Agenda

• Intel® Manycore Platform Software Stack

• Getting Started with Intel® Xeon Phi™

• Demonstration

• References
Software drivers for Intel® Xeon Phi Coprocessor

Linux* 64-bit kernel 2.3.34 (or later)
- Tested with Red Hat Enterprise Linux (RHEL 6.0, 6.1, 6.2, 6.3, and 6.4), and SuSE Linux Enterprise Server (SLES 11 SP1/SP2)

Windows* 7, 8, Server 2008 R2, Server 2012
- Currently in Beta phase
Intel® MPSS

Symmetric Communications Interface (SCIF)
- Direct intra-platform two-point connections via PCIe, e.g. (R)DMA
- Enables programming models (CAO, MYO)

Coprocessor Offload Infrastructure (COI)
- Simplifies asynchronous code/data transfer and remote execution
- API that implements Compiler Assisted Offload (CAO)

Coprocessor Communication Link (CCL)
Leverages SCIF via OFED in order to enable InfiniBand
1. Direct access to InfiniBand HCA out of (co-)processor; low latency
2. Proxy access to pipeline data to InfiniBand network via host memory
Intel MPI dual-DAPL selects data path depending on msg. size and configuration to opt. latency (direct) and bandwidth (direct and proxy).
Don't worry, it's just to illustrate its not a single driver. This is a real Manycore architecture.
Intel® MPSS: Administrative Tools

micflash
updates the flash on the coprocessor; saves and retrieves the version and other information for each section of the flash – POSIX version is mpssflash

miccheck
verifies the configuration of the coprocessor by running various diagnostic tests.

micrasd
logs error conditions and provides automatic restart under some conditions

micinfo
provides information about host and coprocessor system configuration – POSIX version is mpssinfo

micctrl
configures and restarts the coprocessor

micsmc
monitors the coprocessors.

* Services (daemons): mpss, ofed-mic, openibd, opensmd, ...
System Management and Configuration (SMC) micsmc

Settings Dialog

Temperature

Utilization ("top")

Summary

Card 0

Card 1

History
Intel® MPSS: μOS Overview

μOS – Embedded Linux*

- Standard kernel from kernel.org (2.6.38); may be based on Yocto Project for e.g., BitBake package management
- Virtual Ethernet driver, NFS shares, ssh, scp, etc.
- Busybox: minimal shell environment

Open source

- Kernel/OS changes to support Intel® Many Integrated Core Architecture including adjusted GNU* GCC and GDB
- Includes Symmetric Communications Interface (SCIF)

http://software.intel.com/en-us/blogs/2012/06/05/knights-corner-open-source-software-stack
Intel® MPSS: μOS Notes

Copying files to the coprocessor’s file system lowers the amount of available memory

• Virtual file system (ramdisk); better use an NFS share

Commands are by default located at /bin, /usr/bin, /sbin, and /usr/sbin

• Many of the normal commands are actually provided as links from a single executable /bin/busybox

• Visit www.busybox.net to find what commands are available and which options are supported
Agenda

• Intel® Manycore Platform Software Stack
• Getting Started with Intel® Xeon Phi™
• Demonstration
• References
Intel® Compiler

Intel® Compiler for C, C++, and FORTRAN

- C++ (icpc) and C (icc) compiler where C or C++ compilation is selected according to the type of the source file (if not set explicitly)
- Intel® Fortran Compiler (ifort)

Compatibility

- Compatible ABI with platform’s default compiler (gcc)
  - Object files can be mixed between compilers and languages
- Command line options are compatible with respect to functionality that’s common between both compilers
  - Sometimes multiple options for the same feature for historical reasons e.g., -openmp (-fopenmp) or -xHost (-march=native)
Compatibility to Linux GNU Compilers

Compatibility between compilers a topic of four levels:

Compatibility of object code including name mangling

- Code compiled by ICC and GCC can be mixed without exceptions
- Any incompatibility would be considered a very critical bug !!

Compiler switches

- Main switches of GCC and ICC are identical; both have options however not available in the other
- In most cases, gcc in a Makefile simply can be replaced by icc

Source code language – features, syntax and semantic

- for the relevant parts, full compatibility of ICC and GCC – in particular standard conformance (C and C++)

OpenMP: Intel run time works for GCC too

- No restriction to mix OpenMP of ICC- and GCC-compiled code

White paper “Intel® Compilers for Linux*: Compatibility with GNU Compilers” on software.intel.com has all details for all compiler releases
Intel® Compiler and Intel® Xeon Phi™

Intel® Compiler is **three** compilers within a single executable.

1. **Compiler to target Intel Architecture host systems**

2. **Cross-compiler to target Intel MIC Architecture**
   - Use `-mmic` option to request the MIC code path ("native")
   - Compiles any kind of C/C++/F code without restrictions
   - All assumptions about Intel Architecture will hold true

3. **Heterogeneous / offload compiler**
   - Leverages the compiler’s multi-versioning / multi-codepath feature and adds code for code sections marked for offload
   - Three programming models / standards are supported:
     1. Intel Language Extensions for Offload (LEO)
     2. Shared virtual memory model via Intel® Cilk™ Plus, or via Mine-Yours-Ours API (MYO)
     3. OpenMP 4.0 since V13.0.2
Intel® Compiler: Native Target

• Intel MPSS need to be installed and serves as chroot-style environment for cross-compilation

• The term “native” often refers to Manycore hosted execution (coprocessor as an independent node)

• Input data as well as the executable(s) along with dependencies must be made available

• Almost no effort to just get any code running (starting point for optimization)

• Not optimal for sequential phases in the code
Intel® Compiler: Offload

• Offload compilation is automatically enabled when offload directive/pragma is present in code
  - Disable offload compilation: -no-offload (/Qoffload-)

• Compiler runtime takes care for data transfer (offload directives) and code transfer/execution

• Offloaded code section consists of native code; leverages capability for dispatched code paths

• Execution can be further controlled by environment variables on the host; mirrors well-known env.-variables with a user-defined prefix

• Fallback to host code path if no coprocessor is present
  - Check: OFFLOAD_REPORT=level
double A[1000], B[1000], C[1000];
void add() {
    for (int i = 0; i < 1000; ++i) {
        if (A[i] > 0) {
            A[i] += B[i];
        } else {
            A[i] += C[i];
        }
    }
}

.B1.2::
    movaps xmm2, A[rdx*8]
xorps xmm0, xmm0
cmpltpd xmm0, xmm2
movaps xmm1, B[rdx*8]
andps xmm1, xmm0
andnps xmm0, C[rdx*8]
orps xmm1, xmm0
addpd xmm2, xmm1
movaps A[rdx*8], xmm2
add rdx, 2
cmp rdx, 1000
jl .B1.2

.SSE2

.B1.2::
    movaps xmm2, A[rdx*8]
xorps xmm0, xmm0
cmpltpd xmm0, xmm2
movaps xmm1, C[rdx*8]
blendvpd xmm1, B[rdx*8], xmm0
addpd xmm2, xmm1
movaps A[rdx*8], xmm2
add rdx, 2
cmp rdx, 1000
jl .B1.2

.AVX

.B1.2::
    movaps ymm3, A[rdx*8]
vmovaps ymm1, C[rdx*8]
vcmpgtpd ymm2, ymm3, ymm0
vblendvd ymm4, ymm1,B[rdx*8], ymm2
vaddpd ymm5, ymm3, ymm4
vmovaps A[rdx*8], ymm5
add rdx, 4
cmp rdx, 1000
jl .B1.2

.SSE4.1
# Intel® C++ and Fortran Compiler
## Some Options (just a few...)

<table>
<thead>
<tr>
<th>Option</th>
<th>Windows*</th>
<th>Linux*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disable optimization</td>
<td>/Od</td>
<td>-Od</td>
</tr>
<tr>
<td>Optimize for speed (no code size increase)</td>
<td>/O1</td>
<td>-O1</td>
</tr>
<tr>
<td>Optimize for speed (default)</td>
<td>/O2</td>
<td>-O2</td>
</tr>
<tr>
<td>More aggressive code (loop) transformations and optimizations – not necessarily better than -O2</td>
<td>/O3</td>
<td>-O3</td>
</tr>
<tr>
<td>Create symbols for debugging</td>
<td>/Zi</td>
<td>-g</td>
</tr>
<tr>
<td>Inter-procedural optimization</td>
<td>/Qipo</td>
<td>-ipo</td>
</tr>
<tr>
<td>Profile guided optimization (multi-step build)</td>
<td>/Qprof-gen</td>
<td>-prof-gen</td>
</tr>
<tr>
<td></td>
<td>/Qprof-use</td>
<td>-prof-use</td>
</tr>
<tr>
<td>Optimize for speed across the entire program</td>
<td>/fast</td>
<td>-fast</td>
</tr>
<tr>
<td>(not recommended on Linux due to static linking! )</td>
<td>(same as: /O3 /Qipo /Qprec-div -/QxHost)</td>
<td>(same as: -ipo -O3 -no-prec-div -static -xHost)</td>
</tr>
<tr>
<td>OpenMP 3.1 support</td>
<td>/QOpenmp</td>
<td>-openmp</td>
</tr>
<tr>
<td>Automatic parallelization</td>
<td>/Qparallel</td>
<td>-parallel</td>
</tr>
</tbody>
</table>
## Some Intel-Specific Directives

<table>
<thead>
<tr>
<th>Pragma</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>distribute, distribute_point</td>
<td>instructs the compiler to prefer loop distribution at the location indicated</td>
</tr>
<tr>
<td>inline</td>
<td>instructs the compiler that the user prefers that the calls in question be inlined</td>
</tr>
<tr>
<td>ivdep</td>
<td>instructs the compiler to ignore assumed vector dependencies</td>
</tr>
<tr>
<td>loop_count</td>
<td>indicates the loop count is likely to be an integer</td>
</tr>
<tr>
<td>novector</td>
<td>specifies that the loop should never be vectorized</td>
</tr>
<tr>
<td>optimization_level</td>
<td>enables control of optimization for a specific function</td>
</tr>
<tr>
<td>parallel/noparallel</td>
<td>facilitates auto-parallelization of an immediately following DO loop; using keyword [always] forces the compiler to auto-parallelize; noparallel pragma prevents auto-parallelization</td>
</tr>
<tr>
<td>simd</td>
<td>enforces vectorization of innermost loops</td>
</tr>
<tr>
<td>unroll/nounroll</td>
<td>instructs the compiler the number of times to unroll/not to unroll a loop</td>
</tr>
<tr>
<td>unroll_and_jam/nounroll_and_jam</td>
<td>instructs the compiler to partially unroll higher loops and jam the resulting loops back together. Specifying the nounroll_and_jam pragma prevents unrolling and jamming of loops.</td>
</tr>
<tr>
<td>unused</td>
<td>describes variables that are unused (warnings not generated)</td>
</tr>
<tr>
<td>vector</td>
<td>indicates to the compiler that the loop should be vectorized according to the arguments: always/aligned/unaligned/nontemporal/temporal</td>
</tr>
</tbody>
</table>
Compiler switch:

- `opt-report-phase[=phase]` (Linux)
  
  `phase` can be:

  - `ipo` – Interprocedural Optimization
  - `ilo` – Intermediate Language Scalar Optimization
  - `hpo` – High Performance Optimization
  - `hlo` – High-level Optimization

  ...  

  - `all` – All optimizations

Control the level of detail in the report:

- `/Qopt-report[0|1|2|3]` (Windows)
- `-opt-report[0|1|2|3]` (Linux, MacOS X)
Optimization Report Example

```plaintext
icc -O3 -opt-report-phase=hlo -opt-report-phase=hpo
```

```
... LOOP INTERCHANGE in loops at line: 7 8 9
Loopnest permutation ( 1 2 3 ) --> ( 2 3 1 )
...
Loop at line 8 blocked by 128
Loop at line 9 blocked by 128
Loop at line 10 blocked by 128
...
Loop at line 10 unrolled and jammed by 4
Loop at line 8 unrolled and jammed by 4
```
Profile-Guided Optimizations (PGO)

Static analysis leaves many questions open for the optimizer like:

- How often is \( x > y \)
- What is the size of count
- Which code is touched how often

Use execution-time feedback to guide (final) optimization

Enhancements with PGO:

- More accurate branch prediction
- Basic block movement to improve instruction cache behavior
- Better decision of functions to inline (help IPO)
- Can optimize function ordering
- Switch-statement optimization
- Better vectorization decisions

```plaintext
if (x > y)
    do_this();
else
    do_that();

for(i=0; i<count; ++I)
    do_work();
```
PGO Usage: Three Step Process

Step 1
Compile + link to add instrumentation

```
icc -prof_gen foo.c -o foo
```

Step 2
Execute instrumented program

```
foo.exe (on a typical dataset)
```

Dynamic profile:

```
12345678.dyn
```

Merged .dyn files:

```
pgopti.dpi
```

Step 3
Compile + link using feedback

```
icc -prof_use prog.c -o foo
```

Optimized executable:

```
foo.exe
```

Instrumented executable:

```
foo.exe
```

profmerge
for (i=0; i < NUM_BLOCKS; i++)
{
    switch (check3(i))
    {
        case 3:                    /* 25% */
            x[i] = 3;  break;
        case 10:                   /* 75% */
            x[i] = i+10;  break;
        default:                   /* 0% */
            x[i] = 99;  break
    }
}

“Case 10” is moved to the beginning

• PGO can eliminate most tests & jumps for the common case – less branch mispredicts
Interprocedural Optimizations
Extends optimizations across file boundaries

Without IPO

Compile & Optimize -> file1.c
Compile & Optimize -> file2.c
Compile & Optimize -> file3.c
Compile & Optimize -> file4.c

With IPO

Compile & Optimize

/\Qip, -ip Only between modules of one source file
/\Qipo, -ipo Modules of multiple files/whole application
## Interprocedural Optimizations (IPO)

### Usage: Two-Step Process

<table>
<thead>
<tr>
<th>Compiling</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linux</strong>*</td>
<td><code>icc -c -ipo main.c func1.c</code>  &lt;br&gt; <code>func2.c</code></td>
</tr>
<tr>
<td><strong>Windows</strong>*</td>
<td><code>icl -c /Qipo main.c func1.c</code>  &lt;br&gt; <code>func2.c</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Linking</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linux</strong>*</td>
<td><code>icc -ipo main.o func1.o</code>  &lt;br&gt; <code>func2.o</code></td>
</tr>
<tr>
<td><strong>Windows</strong>*</td>
<td><code>icl /Qipo main.o func1.o</code>  &lt;br&gt; <code>func2.obj</code></td>
</tr>
</tbody>
</table>
What you should know about IPO

O2 and O3 activate “almost” file-local IPO (-ip) by default

• Only a very few, time-consuming IP-optimizations are not done but for most codes, -ip is not adding anything

IPO extends compilation time and memory usage

In-lining of functions is most important feature of IPO but there is much more

• Inter-procedural constant propagation
• MOD/REF analysis (for dependence analysis)
• Routine attribute propagation
• Dead code elimination
• Induction variable recognition
• …many, many more

IPO works for (shared and static) libraries too

• Not a trivial topic
Inlining

To inline a routine \( B \) (the callee) into a routine \( A \) (the caller) at a callsite \( C \), we

- Replace the call to \( B \) by the a copy of the code for \( B \)
- Replace the formal parameters of \( B \) by the actual parameters at the callsite \( C \)
- Make each return in the copy of \( B \)'s code branch to the statement following the copy of \( B \)'s code
Inlining Example

Before Inlining:

```fortran
program main
  dimension a(1000)
  call init(a, 1000)
  result = exsum(a, 1000)
  print *, result
end

subroutine init(a, n)
  dimension a(n)
  do i = 1, n
    a(i) = real(i)
  end do
end

real function exsum(a, n)
  dimension a(n)
  exsum = 0
  do i = 1, n
    exsum = exsum + a(i)
  end do
end
```

After Inlining:

```fortran
program main
  dimension a(1000)
  call init(a, 1000)
  exsum = 0
  do i = 1, n
    exsum = exsum + a(i)
  end do
  result = exsum
  print *, result
end

subroutine init(a, n)
  dimension a(n)
  do i = 1, n
    a(i) = real(i)
  end do
end

real function exsum(a, n)
  dimension a(n)
  do i = 1, n
    a(i) = real(i)
  end do
end
```
What are Benefits of Inlining?

Inlining’s benefits make it an almost perfect IP/IPO transformation

- Does not require whole program
- Almost always legal to perform
  - Linux* (System V ABI) defines restrictions for position-independent code (that is shared libraries) / -fpic switch
- Can almost always be used to improve performance
  - Reduced call overhead
  - More code locality
  - Enable other optimizations like vectorization
  - Better dependence analysis
  - Better memory disambiguation

Frequently inlining is the only important (helpful) optimization for IP/IPO

Key issue: Which call-sites to inline?
What are Inlining’s Disadvantages?

Inlining has multiple, negative side effects too

- Overall code size increases
  - Can be orders of magnitude, especially in C++
- Code size of calling routine increases resulting in larger compile time
  - Induces non-linear slowdowns in later compiler transformations
- Can cause register pressure in the calling procedure requiring new memory moves
- Debugging, Performance Analysis (VTune)
  - Procedures disappear – relationship to original source code more difficult
What are inlining heuristics?

A set of rules that specify which callsites should be inlined

Goal is to find right balance between advantages and disadvantages to reach best performance gain

Compiler uses complex rule set which can’t be optimal in all cases

• Frequently it is worth to look into the details
Compiler Reporting for Inlining

Available as part of the opt report

Opt report only for IPO/in-lining

• Windows* syntax
  /Qopt_report /Qopt_report_report_phase ipo

• Linux syntax
  -opt_report -opt_report_report_phase ipo
Example of In-lining Opt Report

```
program main
subroutine init(a, n)
dimension a(n)
call init(a, 1000)
result = exsum(a, 1000)
print *, result
end

subroutine init(a, n)
dimension a(n)
do i = 1, n
    a(i) = real(i)
end do
end

real function exsum(a, n)
dimension a(n)
exsum = 0
do i = 1, n
    exsum = exsum + a(i)
end do
end

INLINING: (_MAIN__)

EXTERN: The code for this routine is not available to compiler
INLINE:  In-lining successfully happened

--> _for_write_seq_lis(EXTERN)
-->  INLINE:  _EXSUM(3) (isz = 15) (sz = 23 (13+10))
-->  _INIT(1) (isz = 14) (sz = 21 (12+9))

INLINE:  In-lining successfully happened
-->  _for_set_reentrancy(EXTERN)
```
Reading the Inlining Report Details

->INLINE: _<name>_(#)(isz)(sz=(s1+s2))

#: unique number of the called routine

<name>: (mangled) name of routine being in-lined

s1: Size of calling function before inl.
  (internal representation, not source lines)

s1: Size of called function

sz: Sum of s1 and s2

isz: Size of calling function after all optimizations ( estimate )

Typically: s1 < isz < sz
Options to Control In-lining

Global control

no-ip-inline, inline-limit=<number>

Keywords for individual routines

inline, __inline, __forceinline

Coarse grain control of limits

inline-factor=<number> or no-inline-factor

Individual control of each limit

Size limit for small routines: -inline-min-size
Size limit for large routines: -inline-max-size
Limit on increase in routine size: -inline-max-total-size
Limit on inlining instances per routine: -inline-max-per-routine
Limit on inlining instances per compile: -inline-max-per-compile

Linux C/C++ attributes (same as for GCC)

attribute ((always_inline))
New since 12.0 Compiler

Pragma to control in-lining of calls in a statement or block of statements; very fine grained control on in-lining

Syntax:

```
#pragma inline[recursive]
#pragma forceinline[recursive]
#pragma noinline
```

These statement (call-site)-specific pragmas take precedence over the corresponding function-specific pragmas which take precedence over compiler switches
Method for getting Best Performance from Inlining

Find hottest call-sites
• E.g. using callgraph feature of performance analyzer like Inte® Vtune Amplifier XE

Determine if these call-sites got inlined
• Using IPO-reporting

Change inlining behavior
• Using options, pragmas, keywords, inlining lists, or script control

Measure difference in performance
Agenda

• Intel® Manycore Platform Software Stack

• Getting Started with Intel® Xeon Phi™

• Demonstration

• References
EURORA: Login

Login node for Eurora cluster (also from ext.)

$ ssh user@login.eurora.cineca.it

Nodes with Intel® Xeon Phi™: node018...node024

$ ssh node018
$ ssh node018-mic1
EURORA: Environment

Compiler and analysis tools

$ module load intel

Compiler  icc, icpc, ifort
Inspector  inspxe-cl, inspxe-gui
Amplifier amplxe-cl, amplxe-gui

Math library

$ module load mkl

Location  $MKLROOT
EURORA: More to Know...

Scratch area to share files

$ echo $CINECA_SCRATCH
  /gpfs/scratch/usera06trn/$USER

Documentation

http://www.hpc.cineca.it/content/eurora-user-guide
**Example: Native Execution**

```
[host] $ source /opt/intel/composerxe/bin/compilervars.sh intel64

[host] $ icc --version
icc (ICC) 13.1.3 20130607
Copyright (C) 1985-2013 Intel Corporation. All rights reserved.

[host] $ icpc -mmic -O2 -fopenmp -DNDEBUG test.cpp -o test

[host] $ scp test host-mic0:/tmp
[host] $ scp /path/to/libiomp5.so host-mic0:/tmp

[host] $ ssh host-mic0

[host-mic0] $ cd /tmp
[host-mic0] $ LD_LIBRARY_PATH=..:$LD_LIBRARY_PATH ./test
```
Activity 1: Native Execution

1. Pick-up the material from the shared/scratch area

2. Make the Intel Compiler available (environment)

3. Cross-compile the example code
   - Consists of two translation units (main.cpp, pi.[cpp|f90])
   - Use a one-step build (optional: separate compile/link)

4. Copy the executable to the coprocessor
   - Run the executable and discover missed dependencies
   - Alternative: $ ldd executable
   - Note: remember to set LD_LIBRARY_PATH

5. Run the program
Activity 2: Offload Execution

1. Pick-up the material from the shared/scratch area

2. Make the Intel Compiler available (environment)

3. Employ the offload directive and decide about what to offload: loop section or pi_kernel function
   - Implement main-offload.cpp (copy of main.cpp)

4. Compile the code in one step
   - Optional: try to offload the Fortran based implementation using the C++ main program

5. Run the program
   - Check whether an offload happened
Activity 3: Parallel Native Execution

1. Pick-up the material from the shared/scratch area
2. Make the Intel Compiler available (environment)
3. Parallelize the loop of pi_kernel using OpenMP*
   - Implement main-parallel.cpp (copy of main.cpp)
4. Cross-compile and run the code
5. How to check whether your parallelization is correct or not?
Agenda

• Intel® Manycore Platform Software Stack
• Getting Started with Intel® Xeon Phi™
• Demonstration
• References
References

Intel® Xeon Phi™ Coprocessor Developer Forum
http://software.intel.com/mic-developer

Intel® Xeon Phi™ Coprocessor Quick Start Guide

Programming Intel's Xeon Phi: A Jumpstart Introduction
http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

Phi Programming for CUDA Developers
Available since March 2013

Completely focused on Intel Xeon Phi coprocessors.

Volume 1: essentials

~380 pages of explanation of programming.

Teaches us how to use and obtain high performance on the Intel MIC architecture

“The authors have provided a very readable, big-picture introduction to programming the Intel Xeon Phi Coprocessor. By chronicling step-by-step optimizations of several computational kernels, software interfaces are illustrated for getting the most out of key architectural features of the Intel Xeon Phi Coprocessor.”

James L. Schwarzmeier, Cray Inc, January 2013

(c) 2013, Morgan Kaufman Publ. Inc
INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY
ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS
DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR
IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES
RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY
PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Software and workloads used in performance tests may have been optimized for performance only on
Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using
specific computer systems, components, software, operations and functions. Any change to any of
those factors may cause the results to vary. You should consult other information and performance
tests to assist you in fully evaluating your contemplated purchases, including the performance of that
product when combined with other products.

Copyright © 2013, Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core,
VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

---

**Optimization Notice**

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that
are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and
other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on
microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended
for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for
Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information
regarding the specific instruction sets covered by this notice.

Notice revision #20110804